Key functions and neural network with tensor sequential

```
  mean = np.mean(data, axis=0)
  std_dev = np.std(data, axis=0)

  # reenforce the division not 0
  std_dev[std_dev == 0] = 1

  #  x = (x-mean) / std
  data = (data - mean) / std_dev
  return data

# Convert labels to one-hot encoding https://www.tensorflow.org/api_docs/python/tf/one_hot
def preprocessY(labels):
  return tf.one_hot(labels, depth=10)

# build model
def tf_NeuralNetwork(n_neurons, dropout=1):
  model = tf.keras.models.Sequential([
      Flatten(input_shape=(784,)), # match with reshaped datapints dimension
      Dense(n_neurons, activation='relu', kernel_initializer='he_normal'), # first layer with
      Dropout(dropout),
      Dense(10, activation='softmax')
  ])
  return model

# randomize the targets
def noisyY(labels, noise_ratio):
  label_copy = labels.copy()  # it is nessary to modify on a copy dataset, other wise this fur
  num_noisy = int(len(labels) * noise_ratio)
  noisy_indices = np.random.choice(len(labels), num_noisy, replace=False) #generate an array w

  for i in noisy_indices:
      label_copy[i] = np.random.randint(0, 10)
  #label_copy[0] = 10
  return label_copy
```

Question 2

Implementations to train with original dataset

```
# Parameters
width_k = [1, 5, 15, 35]
dropout_p = [0.9, 0.75, 0.5, 0]
epoch = 80
batch_size = 100

# Training models and storing final results in a list of dictionaries (original dataset)
results = []

for p in dropout_p:
  kth_train = []
  kth_val = []
  for k in width_k:
    model = tf_NeuralNetwork(n_neurons=k, dropout=p) # it create a 2layer neural network tensc

    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy']) #ir

    history = model.fit(x_train, y_train, epochs=epoch, batch_size=batch_size, validation_data

    final_train_acc = history.history['accuracy'][-1]

    final_val_acc = history.history['val_accuracy'][-1]

    #save as dictionary easy to extract
    results.append({'k': k, 'p': p, 'train_acc': final_train_acc, 'val_acc': final_val_acc})

    print(p, k, " th training done")
```

Orignal dataset plot

```
# Plot training accuracy
for p in dropout_p:
  train_accuracies = [result['train_acc'] for result in results if result['p'] == p]
  plt.plot(width_k, train_accuracies, label="p =" +str(p))

plt.xlabel('n_neurons')
plt.ylabel('Training Acc')
plt.title('Training Accuracy vs #neurons with difference dropout(Original Dataset)')
plt.legend()
plt.show()


# Plot test accuracy
for p in dropout_p:
  test_accuracies = [result['val_acc'] for result in results if result['p'] == p]
  plt.plot(width_k, test_accuracies, label="p =" +str(p))

plt.xlabel('n_neurons)')
plt.ylabel('Test Acc')
plt.title('Test Accuracy vs #neurons with difference dropout(Original Dataset)')
plt.legend()
plt.show()
```

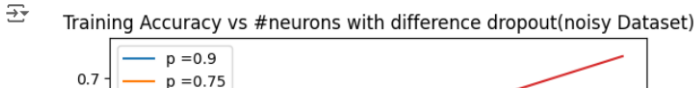
Question3

Noislized Y targeet. Based on the result, the final accuracy is interesting. We will track the losses on each epochs too

```
# Training models and storing final results (noisy dataset)
noisy_results = []
noisy_fit_results = []
for p in dropout_p:
  for k in width_k:
    model = tf_NeuralNetwork(n_neurons=k, dropout=p)

    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

    history = model.fit(x_train, y_train_noisy, epochs=epoch, batch_size=batch_size, validati


    # Track the loss on each epoch
    fit_train_acc = history.history['accuracy']
    fit_val_acc = history.history['val_accuracy']
    fit_loss = history.history['loss']
    noisy_fit_results.append({'k': k, 'p': p, 'fit_train__acc': fit_train_acc, 'fit_val_acc':

    # Track the final result
    final_train_acc = history.history['accuracy'][-1]
    final_val_acc = history.history['val_accuracy'][-1]
    noisy_results.append({'k': k, 'p': p, 'train_acc': final_train_acc, 'val_acc': final_val_

    print(p, k, " th training done")
```

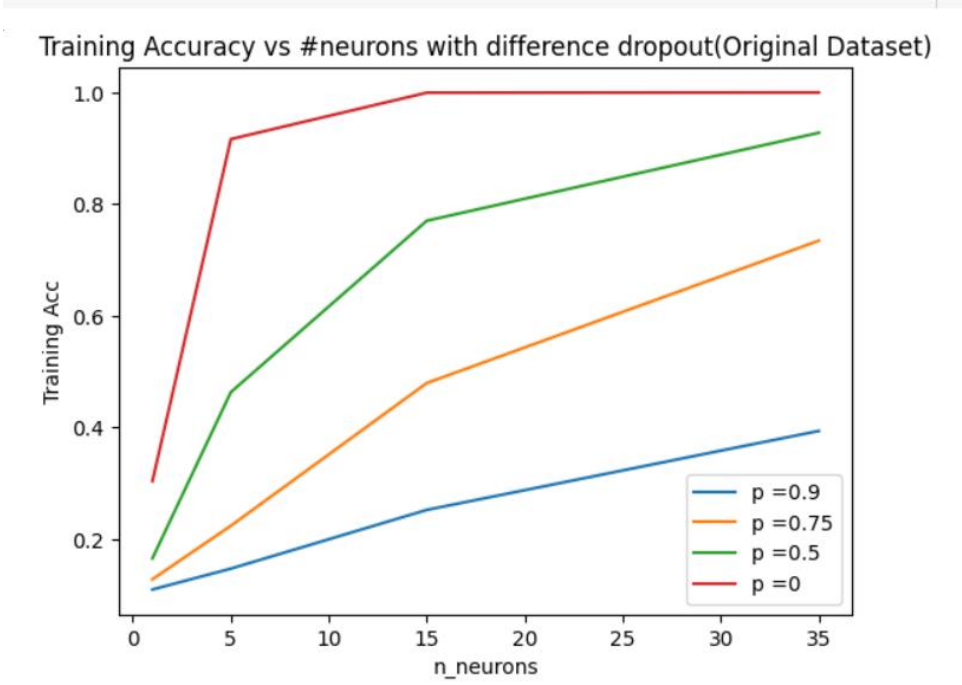
Noisy dataset plot

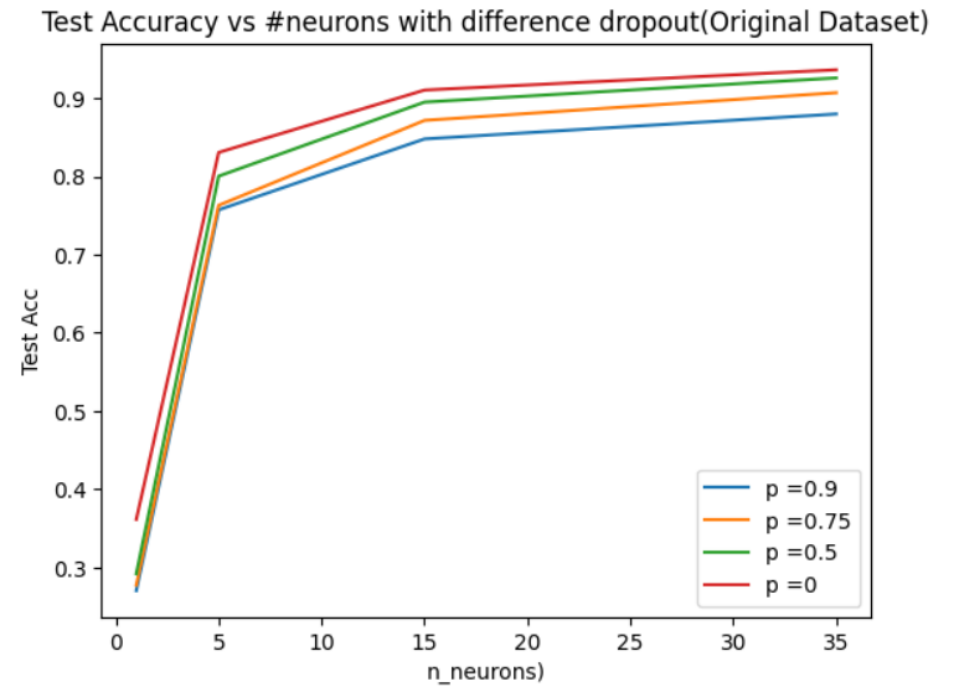
```
# Plot training accuracy
for p in dropout_p:
  train_accuracies = [result['train_acc'] for result in noisy_results if result['p'] == p]
  plt.plot(width_k, train_accuracies, label="p =" +str(p))

plt.xlabel('n_neurons')
plt.ylabel('Training Acc')
plt.title('Training Accuracy vs #neurons with difference dropout(noisy Dataset)')
plt.legend()
plt.show()


# Plot test accuracy
for p in dropout_p:
  test_accuracies = [result['val_acc'] for result in noisy_results if result['p'] == p]
  plt.plot(width_k, test_accuracies, label="p =" +str(p))

plt.xlabel('n_neurons')
plt.ylabel('Test Acc')
plt.title('Test Accuracy vs #neurons with difference dropout(noisy Dataset)')
plt.legend()
plt.show()
```



Comment on the differences between Step 2 and Step 3. How does noise change things? For which setup dropout is more useful?

For the original dataset

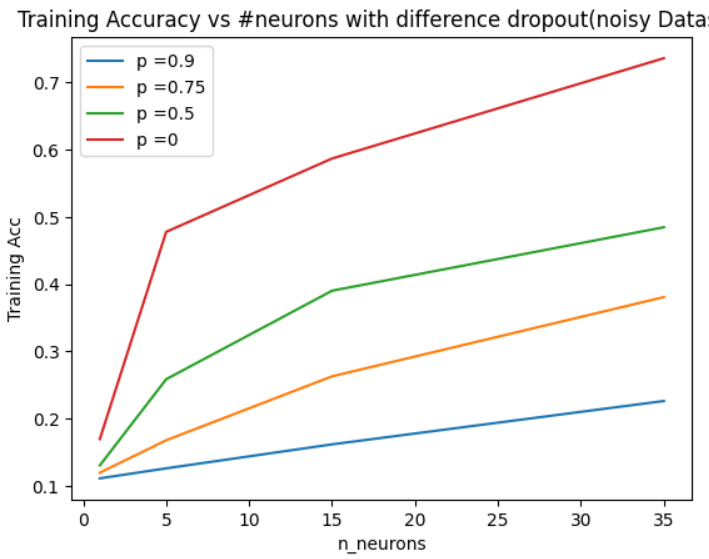


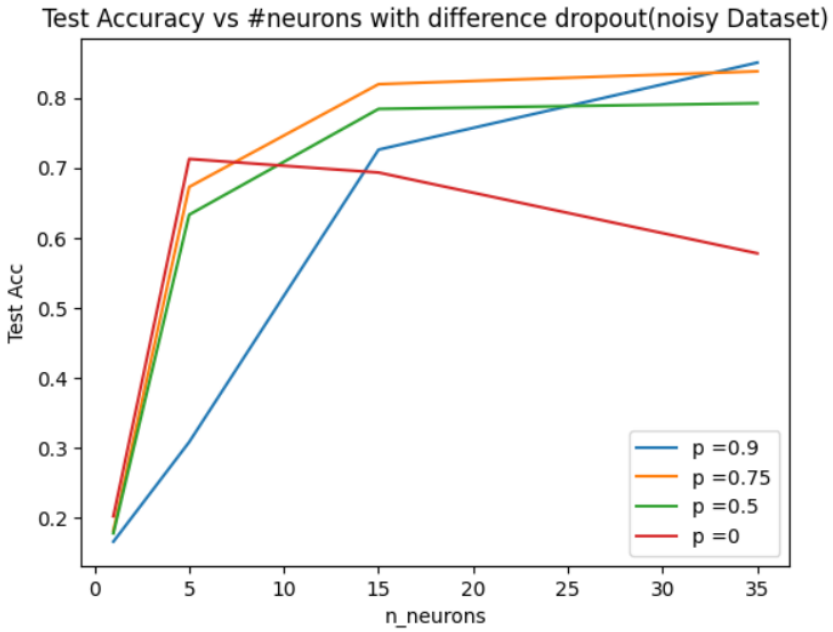


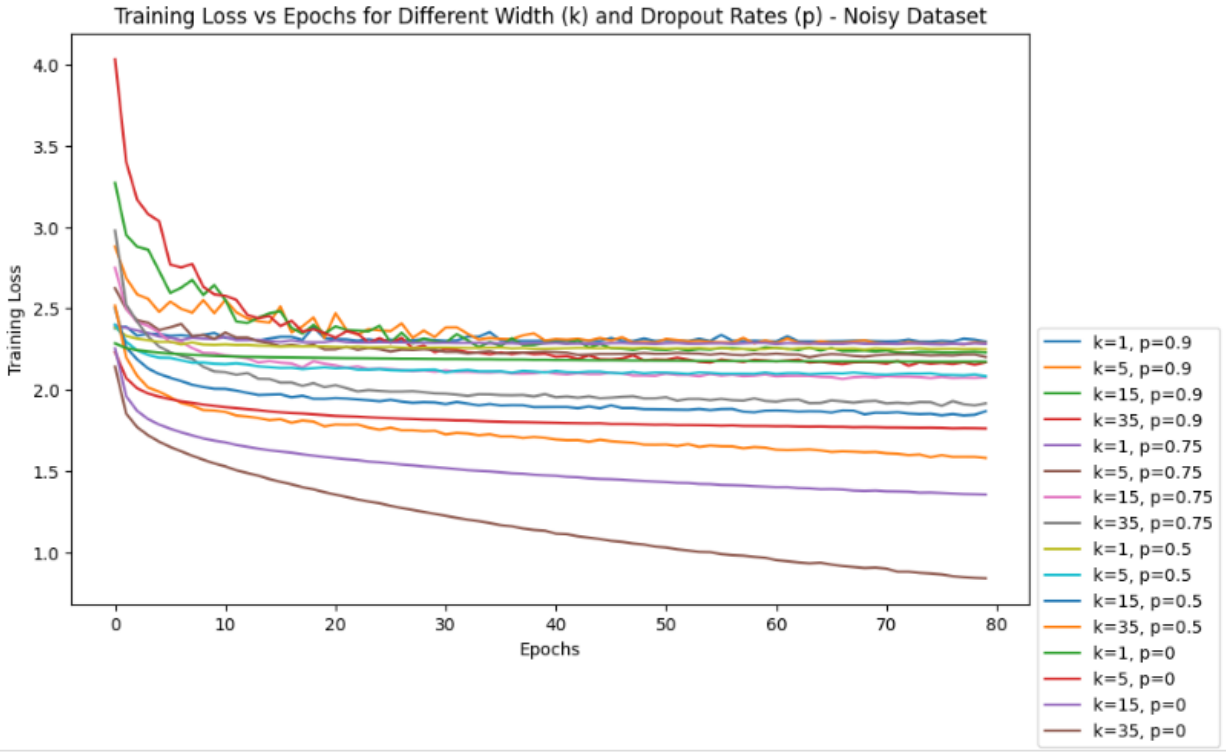
On training plot, we observe that the accuracy increases as both the number of neurons (k) and dropout rate (p) increase. Training with a dropout rate of p = 0 (no neurons are turned off, all neurons are used) and total neurons at k = 35 yields the highest training accuracy, which is almost 100%. Referring to lecture and theory, this is a sign of overfitting. To prove that we can look at the test accuracy plot, where the accuracy for p = 0, k = 35 drops from 99.9% to 90% compared to the training accuracy.

While the accuracy for dropout p = 0.5 and 0.75 remains stable in the test accuracy plot. And dropout p = 0.5 has higher test accuracy than p =0.75. Additionally, In this case, the neural network is more sensitive to the dropout rate than to the number of neurons. This is evident because, as the number of neurons increases, the test accuracy stabilizes around approximately 0.9, indicating that the model is learning effectively without overfitting.

For the noisy dataset


Training Accuracy vs #neurons with difference dropout(noisy Dataset)
p =0.9
p =0.75
p =0.5
p =0
Training Acc
n_neurons
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
5
10
15
20
25
30
35


Test Accuracy vs #neurons with difference dropout(noisy Dataset)
Test Acc
n_neurons
p =0.9
p =0.75
p =0.5
p =0
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
5
10
15
20
25
30
35



First take look the test and train the accuracy plot for noisy dataset, the plot for p = 0 and k = 35 is eye attractive. We see that while the training accuracy increases, the test accuracy drops significantly. This makes sense because, based on our observations from the original dataset, the model is overfitting, which implies it is very sensitive all features. Since we randomize the target to fool the model, the model try to process features with noisy labels, which results in increasing in training set and decreasing during testing.

For the rest of the models with different hyperparameter, their performance also align with the observation from the original dataset. For p=0.75 and 5, both models do not overfit and are less sensitive to individual features. This means that, even with mislabeled targets, these models are less effective at updating their weights based on incorrect data, the optimizers are regulated. For the model with p = 0.1, there isn't a significant difference in training set since this model is underfitting and not learning effectively from the training samples. However, the regulation grants the ability of anti-poison data. It accuracy increases on the test set.

For the loss over epochs, All models showed a decreasing in loss, with the k=35, p=0 which is the most overfitting model stands out in the training set on minimizing the loss. In general,the lower in p the lower in loss which aligns with previous observations. Models with more neurons turned on are more sensitive to features.

Conclusion:

The accuracy of the models is more sensitive on dropout rate than to the number of neurons. This sensitivity is critical especially when the dataset is noisy or not very clean. When the dropout rate is very high (i.e., not turning off at least 5% of the neurons), even a small portion of mislabeled data can be lethal to the model's performance. I believe that a dropout rate of $p = 0.75$ balance between bias and variances. If you are unsure about the integrity of your dataset (e.g., if the dataset contains noise), using a dropout rate of $p = 0.5$ could be a good choice to ensure the model's generalization and avoid overfitting.